# 如何优雅的执行任务-时序并发 副本

👀 更多做任务系列

- 如何优雅的执行任务-时序并发 （本文）
- 如何优雅的执行任务-并发有序

## TL; DR

💡 又来执行任务了，这次的任务场景是和**并发控制**相关

想象一下报社里的新闻编辑，每次优先处理最新的新闻，但报社的人手有限，所以当有新任务时，老的新闻就会被抛弃

## 任务抽象

这里的并发模型可以抽象成：有 n 个 task processor，意味着最多同时处理 n 个任务，当新的任务需要处理，会抛弃最早执行的任务

**伪代码说明：**

```
const MAX_PROCESSOR = 3;
const runner = maxConcurrencyRunner(MAX_PROCESSOR);
const running1 = runner(task1);
```

```
const running2 = runner(task2);
const running3 = runner(task3);

const running4 = runner(task4); // will cancel task1

const results = await Promise.all([
    result1,
    running2,
    running3,
    running4,
])
// results[0] -> 'canceled'
```

## 思路 & 代码实现

### 实现思路

1. 将任务执行做一层包裹
2. 记录当前任务的 id
3. 检查是否有前序任务需要取消，如果有，则标记那个任务 id 为取消状态
4. 任务执行后检查是否被取消，取消则抛出异常

### 适用场景

时序并发控制的适用场景：

- 有资源数限制，每次异步任务的 handler 需要分配有限的计算资源（UDP？）
- 接口轮询/事件/交互冲突，优先处理最新触发的事件（通常最大并发数为 1）
- ...

### 最终版代码

```
export enum TaskResult {
    CANCEL = '__cancel__',
}

/**
 * 最大并发控制 允许 limit 个并发的任务
 * 新增的第 n 个任务后 会将第 n - limit 个任务取消: throw TaskResult.CANCEL
 * @param limit 最大任务数
```

```
 * @returns
 * @example
 * const limitRequestGet = maxConcurrencyScheduler(1)(requestGet);
 * limitRequestGet(); // -> promise throw TaskResult.CANCEL
 * limitRequestGet(); // -> promise throw TaskResult.CANCEL
 * limitRequestGet(); // -> promise settled!
 */
export const maxConcurrencyScheduler =
    (limit = 1) =>
    <Args extends unknown[], R>(task: (...args: Args) => Promise<R>, onCancel?: () => void) => {
        let current = 0;
        const taskMap = new Map<number, boolean>(); // <taskId, taskShouldBeFinished?>

        const cancelTask = (taskId: number) => {
            taskMap.set(taskId, false);
            onCancel?.();
        };
        const isCanceled = (taskId: number) => !taskMap.get(taskId);
        const checkCanceled = (taskId: number) => {
            const canceled = isCanceled(taskId);
            taskMap.delete(taskId);
            if (canceled) {
                throw TaskResult.CANCEL;
            }
        };

        return async (...args: Args): Promise<R> => {
            const taskId = current + 1; // increment id
            if (taskMap.has(taskId - limit)) {
                cancelTask(taskId - limit);
            }
            current = taskId;
            taskMap.set(taskId, true);
            try {
                const p = await task(...args);
                checkCanceled(taskId);
                return p;
            } catch (err) {
                checkCanceled(taskId); // throw cancel flag if err
                throw err;
            }
        };
    };

```

使用 demo 🌰

**例：请求 API 时，防止新老请求冲突数据，仅处理最新的一个任务**

```ts
// 防止新老请求冲突数据 仅处理最新的一个任务
const limitedGetSth = maxConcurrencyScheduler(1)(requestSth);

// 多个无法预测的交互场景会 getData
const getData = async (...args): Promise<{
    silent?: boolean;
    result?: any;
}> => {
    try {
        const res = await limitedGetSth(..args);
        //...
        return {
            result: res,
        }
    } catch (err) {
        if (err === TaskResult.CANCEL) {
            // 被并发控制取消了 无任何操作
            return {
                silent: true,  // 返回 silent 无需处理
            };
        }
        throw err;
    }
}

// 场景1
// getData().then(res => { if (res.silent) return; ... })
// 场景2
// getData().then(res => { if (res.silent) return; ... })
// 场景3
// getData().then(res => { if (res.silent) return; ... })
```